Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 126 – 135

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Effective Implementation of DGEMM on modern multicore CPU

Pawel Gepner*, Victor Gamayunov, David L. Fraser

Intel Corporation,Pipers Way,Swindon Wiltshire SN3 1RJ, United Kingdom

Abstract

In this paper we will present a detailed study on tuning double-precision matrix-matrix multiplication (DGEMM) on the Intel Xeon E5-2680 CPU. We selected an optimal algorithm from the instruction set perspective as well software tools optimized for Intel Advance Vector Extensions (AVX). Our optimizations included the use of vector memory operations, and AVX instructions. Our proposed algorithm achieves a performance improvement of 33% compared to the latest results achieved using the Intel Math Kernel Library DGEMM subroutine.

Keywords: AVX; performance; HPC; Intel

1. Introduction

Dense matrix operations are an important element in scientific and engineering computing applications. Today we have a lot of high performance libraries for dense matrix calculations. Basic Linear Algebra Subprograms (BLAS) is a standard application programming interface to execute basic linear algebra operations such as vector and matrix multiplication [1]. Historically the first BLAS was released as a building block of LAPACK [2], which is a performance portable library for implementing dense linear algebra. Many of the hardware vendors also provide BLAS libraries tuned on their own architecture, i.e. Intel MKL, AMD CML or CUBLAS for the NVIDIA GPUs. To get the best performance all vendors optimize BLAS code specifically to the underlying hardware [2, 3].

Over time optimizations for new CPUs have improved and through the use of new architectural features performance is significantly increasing. A new version of Intel MKL DGEMM subroutine utilizes new architecture extensions and uses AVX extensively, significantly improving the achieved performance versus results delivered by Intel MKL version limited to SSE 4.2 only [4].

The Intel Xeon E5-2680 CPU has been designated for the server and workstation market but many of the architecture features are identical to the 2nd generation Intel Core processor family products. AVX is the most important technology implemented to increase performance for floating-point intensive code. A new memory controller, integrated PCIe 3.0 as well as a faster QPI interface improve overall platform performance and the new capabilities are broadly applicable for HPC applications.

We will run the new method of matrix multiplication code to evaluate the performance of the new processor in comparison with Intel Math Kernel Library DGEMM subroutine.

* Corresponding author. Tel.: +48602414128; fax: +48225708122.
E-mail address: pawel.gepner@intel.com.

doi:10.1016/j.procs.2012.04.014

To answer the research questions, we have organized the paper in the following way. Section 2 presents the platform architecture of the system, introduces AVX and deeply describes the microarchitecture of Intel Xeon E5-2680. In section 3 we characterize matrix multiplication algorithms. In section 4 we evaluate two types of algorithms for dense matrix-multiplications and analyze the performance considerations. In section 5 we discuss the effective hybrid method for dense matrix-multiplications. In section 6 we draw conclusions and outline future work.

2. Platform Architecture and System Configuration

In our study we utilized dual socket server systems: based on Intel Xeon E5-2680 CPU 2.7GHz processors. This platform codenamed Romley-EP represents typical high-end server configurations. The tested platform runs compiled versions of binaries, with AVX support, and linked against the AVX version of the Intel MKL library.

The tested platform is based on the preproduction Intel server board S2600CP "Canoe Pass" with 128GB system memory (16x8GB DDR3, 1333MHz) populated in 16 DIMM slots. We also deployed enterprise class 1TB SATA hard disk drives. The operating system installed is Red Hat Enterprise Linux Server release 6.1, kernel 2.6.32-131.0.15.el6.x86_64. The new version of the Intel software tool kit has been also been installed. The Intel Composer XE 2011 includes Intel Compiler 12.1 as well as the Intel Math Kernel Library (MKL) 10.3. The new compiler and libraries offer advanced vectorization support, including support for Intel AVX and Intel Parallel Building Blocks (PBB), OpenMP, High-Performance Parallel Optimizer (HPO), Interprocedural Optimization (IPO) and Profile-Guided Optimization (PGO). All performance tools and libraries provide optimized parallel functions and data processing routines for high-performance applications and additionally contain several enhancements, including improved Intel AVX support.

2.1. Intel Advanced Vector Extensions Overview

Intel Advanced Vector Extensions expand the capabilities and the programming environment that was introduced with Streaming SIMD (single-instruction, multiple-data) Extensions in 1999. Intel AVX is focused on the vector floating-point performance in scientific and engineering numerical applications, visual processing, cryptography and other application areas [5]. The primary focus of Intel AVX is to provide an easy, efficient implementation of applications with changeable degrees of thread parallelism, and data vector lengths. Intel AVX offers a significant increase in floating-point performance over previous generations of 128-bit SIMD instruction set extensions. Intel AVX increases the width of the 16 XMM registers from 128 bits to 256 bits. Each register can hold eight single-precision (SP) floating-point values or four double-precision (DP) floating-point values that can be operated on in parallel using SIMD instructions. Intel AVX adds three-operand syntax format (c=a+b), whereas previous instructions could only use two-operand syntax format (a=a+b). This new three-operand syntax improves instruction programming flexibility and efficient encoding of new instruction extensions. Intel AVX also establishes a foundation for future evolution in both instruction set functionality and vector lengths by introducing an efficient instruction encoding scheme, three and four operand instruction syntax, FMA (fused multiply-add) extension and supporting load and store masking.

2.2. Processor characteristic

The Intel Xeon E5- 2680CPU has been developed by two design groups one responsible for the unified single core of the CPU for all segments like server, desktop, mobile, the second team known as uncore has been focused on special features and functionality specific for dedicated segment. Apart from the AVX implementation the core team focused on the major processor core enhancements for example improvements on the decoding phase, enhancements to data cache structure and increased instruction level parallelism (ILP). Sandy Bridge single core has 32KB 8-way associative L1 instruction cache and during the decoding phase it is able to decode four instructions per cycle, converting four X86 instructions to micro-ops. To reduce decoding time the new L0 instruction cache for micro-ops has been added to keep 1.5K micro-ops previously decoded. All micro-ops which are already in the L0 cache, do not need to be fetched, decoded and converted to micro-ops again. Consequently the new L0 cache reduces the power used for these complex tasks by about 80% and improves performance by decreasing decoding latency. A

new feature which improves the data cache performance is the extra load/store port. By adding a second load/store port Sandy Bridge can handle two loads plus one store per cycle automatically doubling the load bandwidth. This change also helps support the AVX instructions, enabling the cache to perform one 256-bit AVX load per cycle and generally increase core internal memory bandwidth from 32 bytes per cycle (16 bytes load 16 bytes store per cycle in previous generation Core based processors) to 48 bytes per cycle (two loads plus one store per cycle). One key architecture goal was to increase the performance by finding more instruction level parallelism. To improve the Out Of Order execution segment and tune the ILP performance required increasing the depth and width of execution units, larger buffers, more data storage, more data movement, and consequently more power and more transistors. Implementing AVX doubles the operand size and generates a need for bigger buffers. One of the methods to achieve the bigger dataflow window is achieved by implementing Physical Register File (PRF) instead of centralized Retirement Register File. This method implements a rename register file that is much larger than the logical register file that decreases data duplication and data transfers and finally eliminates movement of data after calculation. This implementation also increases the size of the scheduler and the number of reorder buffer (ROB) entries by 50% and 30%, respectively and as a result increases the size of the dataflow window globally [5,6].

All the modifications made to the Sandy Bridge cores correspond to changes made in the uncore part of the new CPU. The uncore part integrates last level cache (LLC), the system agent, the memory controller with up to 4 DDR3 memory channels, PCI Express interface generation 3.0, up to 2 QPI links and the DMI. To make this architecture more efficient Sandy Bridge implements a ring interconnect to connect all these parts that enables faster communication between the LLC and the system agent area. The ring interconnect is built out of 4 separate rings, a 32 byte data ring, a request ring, an acknowledge ring and a snoop ring. These rings are fully pipelined and run at the core frequency and voltage so that bandwidth, latency and power scale up and down according to the core. The massive ring wires are routed over the LLC in a way that has no area impact. Access on the ring (core to cache, cache to core or cache to system agent) always takes the shortest path. The arbitration on the ring is distributed meaning each ring stops doing its own arbitration. This is a complex architecture that has special sending rules.

Sandy Bridge divides the LLC cache into 2MB blocks, in total 20MB LLC for 8 core server parts. The LLC contains the interface logic from core to the ring, between the cache controller and the ring. It also implements the ring logic, the local arbitration, and the cache controller itself. Each of the caches contain a full cache pipeline. The cache maintains coherency and ordering for the addresses that are mapped to it. It also maintains "core valid bits" like the previous generation of Intel Core processors to reduce unnecessary snoops. The LLC runs at core frequency and voltage, scaling with the ring and core.

A lot of the functionality found previously in the "North Bridge" has been integrated to the monolithic die of Sandy Bridge. This includes PCIe ports, DMI link, DDR3 memory controller; this section also contains the power control unit. The power control unit is a programmable microcontroller that handles all power management and reset functions for the entire chip. This part is also closely integrated with the ring, providing fast access and high bandwidth to memory and I/O. Coherency between the I/O segment and the main cache is also handled in this section of the chip [7,8].

3. Characteristic of Matrix Multiplications Algorithms

Matrix multiplication defines C=A*B, where A, B and C are m *x* k, k *x* n, m *x* n matrices, respectively. A straightforward implementation of matrix multiplications is three nested loops. This Cauchy brute-force algorithm requires n^3 multiplications operations [9].

More effective algorithms e.g. the Strassen algorithm reduces number of operations to order of $n^{2.8}$ in its place. This method is based on the concept of recursively partitioning a matrix into smaller blocks and reduces the most expensive multiplication operations. The Strassen algorithm is limited for matrixes where each size of the matrix must be a power of 2. For the product of C = A*B, where A, B, C have size of 2^n x 2^n, we partition the matrices into equally sized block matrices:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

where: A_{ij} , B_{ij} , C_{ij} are sub-matrices with a size of 2^{n-1} *x* $2^{n-1}$and then we define new matrices.

$$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$
$$M_2 = (A_{2,1} + A_{2,2})B_{1,1}$$
$$M_3 = A_{1,1}(B_{1,2} - B_{2,2})$$
$$M_4 = A_{2,2}(B_{2,1} - B_{1,1})$$
$$M_5 = (A_{1,1} + A_{1,2})B_{2,2}$$
$$M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$
$$M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

Which are then used to define the matrices $C_{i,j}$. Using the new matrices we can eliminate one matrix multiplication and definite the $C_{i,j}$:

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$

$$C_{1,2} = M_3 + M_5$$

$$C_{2,1} = M_2 + M_4$$

$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

We repeat this process n times until the sub-matrices degenerate into defined thresholds. Practical implementations of Strassen's algorithm switch to standard methods of matrix multiplication for small enough sub-matrices, for which those algorithms are more efficient [10].

BLAS definition of DGEMM takes three matrices (A, B, C) and updates C with *beta**C + *alpha**op(A)*op(B) where *alpha* and *beta* are scalars and op(A), op(B) means that one can insert the transpose, conjugate transpose, or just the matrix as is. In our case we can ignore the op(A), op(B), *alpha*, *beta* and just focus on C = A*B, based on the assumption that if we can do this operation fast we can incorporate a transpose or a scalar update equally as fast.

3.1. Intel AVX and DGEMM

Intel AVX doubles the register width of the Intel SSE 4.2 registers. DGEMM kernels within Intel MKL are mostly made up of three instructions:

- **vmovaps**
- **vaddp**
- **vmulpd**

Another instruction that is useful is vbroadcastsd. This can be used to load the same value into all four 64-bit locations of the Intel AVX registers simultaneously. Intel AVX doubles the theoretical floating-point bandwidth as we can execute a vaddpd and a vmulpd in the same cycle. However, since the register size is twice as big, this means that the theoretical floating-point rate doubles versus SSE 4.2. Of course the register movement rate from the vmovaps is also doubled. With AVX we can process 512 bits of data per cycle with a simultaneous vaddpd and vmulpd. As mentioned, the new Intel AVX registers can hold four 64-bit floating-point values. If we have one register that loads from A, one register that loads from B, and one register that accumulates 4 dot-products simultaneously, the code might look something like:

1. **vmovapd ymm0, [rax]**
2. **vbroadcastsd ymm1, [rbx]**
3. **vmulpd ymm0, ymm0, ymm1**
4. **vaddpd ymm2, ymm2, vmm0**

In this case, let's assume rax contains the address of A, and rbx contains the address of B. We can use one register for A values (ymm0), one register for B values (ymm1), and one register for C values (ymm2). If rax and rbx are incremented appropriately and a loop count is created, at the end of the loop one will have completed four rows of A*B. This code is not register blocked, and the problem with this code is that there are 2 loads per pair of adds and multiplies. The adds and multiplies can execute together in a single cycle, but the resulting loop will be load bound because the loads of A and B will require more than a single cycle [11,12].

4. Evaluation of Matrix Multiplications Algorithms

In this paper we will use the n *x* n square matrices multiplications as the target problem and all tested sampled matrices will have a size of power of 2 as Strassen algorithm requirements. Cauchy and Intel MKL DGEMM do not have these limitations. All the elements of the matrices are randomly generated as double precisions numbers.

We have implemented two algorithms and test them against different sizes of matrices then compare to results achieved with the Intel MKL DGEMM. All the implemented methods are multithreaded. We utilize all 16 available threads.

We also tested the accuracy of the algorithms and monitored usage of the system resources. We counted the number of executed floating-point operations to analyse the resource usage.

For the Strassen algorithm we evaluated the size for partitioned sub-matrixes to find the optimal value where the next recursion step will not improve execution time.

In our study we have been evaluating 6 sizes of matrices 1024 *x* 1024; 2048 *x* 2048; 4096 *x* 4096, 8192 *x* 8192, 16384 *x* 16384, 32768 *x* 32768. We have been compiling only AVX versions of binaries to achieve the best possible results. As indicated by previous work [4] AVX improves performance of DGEMM significantly. In our implementation wherever possible, data is used directly from the source or destination matrix to avoid unnecessary memory copy.

For all tested matrices the allocation of memory for the Cauchy algorithm implementation and Intel MKL was almost half of the memory used by the Strassen algorithm. Table 1 presents the memory allocation for all tested size of matrices calculated by Cauchy, Strassen and Intel MKL.

Table 1. Memory allocation in MB

Size of matrix	Cauchy	Strassen	Intel MKL
1024x1024	24	45.5	24
2048x2048	96	184.2	96
4096x4096	384	752.6	384
8192x8192	1536	2998.5	1536
16384x16384	6144	11780	6144
32768x32768	24576	47515.2	24576

The Strassen algorithm implementation requires a 90%-99% larger memory allocation versus Cauchy and Intel MKL. Implementations of classic brute force algorithm and Intel DGEMM MKL subroutine have the same memory conditions.

Table 2 shows the execution time for all tested size of matrices calculated by Cauchy, Strassen and Intel MKL. The Strassen's algorithm performs better than the traditional matrix multiplication algorithm due to the reduced number of multiplications and better memory separation. However, it requires a large amount of memory to run. Using Intel MKL DGEMM is the most efficient and has same memory requirements as the Cauchy algorithm. The Strassen algorithm is applied recursively. The recursion is continued until the sub-matrixes are optimal and then classic brut-force algorithm is applied. This means that once sub-matrices achieved the threshold the next recursion is not performing and sub-matrices are calculated using the brute-force algorithm. For the validated matrices we tested different thresholds for sub-matrices and observe the implication for execution time. Based on the achieved results we selected the optimal threshold size where execution time is the best. In the Table 2 execution time for Straseen algorithm is presented for the optimal threshold.

Table 2. Execution time in seconds for all tested algorithms

Size of matrix	Cauchy -16 threads	Strassen-16 threads	Intel MKL-16 threads
1024x1024	0.48	0.27	0.067
2048x2048	0.79	0.43	0.119
4096x4096	6.3	3.1	0.875
8192x8192	61	32.4	7.9
16384x16384	449.2	224.9	29.7
32768x32768	4480.9	1604.3	233.7

Intel MKL DGEMM subroutine especially with AVX support performs over 7-19 times better versus classical brute–force algorithm illustrating the potential of the new instruction set and efficient usage of processor caches as well as the superior performance of Intel MKL. As indicated by previous work [4] SSE 4.2 version of Intel MKL can improve performance versus Cauchy close to 350%.

Results achieved with the Strassen algorithm implementation are weaker than Intel MKL DGEMM by 360%-680% depending on the size of matrices, but outperforming the Cauchy brute-force algorithms by 70%-270%.

In Table 3 we have posted a number of floating-point MUL and ADD operations for all 3 algorithms. For the Strassen implementation we show results for the optimal threshold. Our results show that the optimal threshold for sub-matrices is 256. This means that once sub-matrices achieved the threshold the next recursion is not performing and sub-matrices are calculated using brute-force algorithm. For this 16 thread version ideal threshold is different than a single thread version which was generating the best results for sub-matrices size 64 or 128. All the results in Table 3 are measured in Mega (2^{20}) floating point multiplications (FP MUL) and Mega additions (FP ADD).

Table 3. Number of the Mega floating-point MUL and ADD operations for all 3 algorithms

Size of matrix	Cauchy -16 threads FP ADD	Strassen -16 threads FP ADD	Intel MKL-16 threads FP ADD	Cauchy -16 threads FP MUL	Strassen-16 threads FP MUL	Intel MKL-16 threads FP MUL
1024x1024	1024	796	1024	1024	784	1024
2048x2048	8192	5593	8192	8192	5488	8192
4096x4096	65536	39220	65536	65536	38416	65536
8192x8192	524288	274831	524288	524288	268912	524288
16384x16384	4194304	1924966	4194304	4194304	1882384	4194304
32768x32768	33554432	13479373	33554432	33554432	13176688	33554432

For Cauchy and the Intel MKL DGEMM algorithms the number of floating-point MUL and ADD operations is the same as for the Strassen algorithms and is dependent on the size of the threshold sub-matrix. For the Strassen optimal thresholds are reported in Table 3, the number of floating-point ADD operations is 28%-148% smaller than floating-point ADD operations for Cauchy and Intel MKL DGEMM subroutine.

The floating-point MUL operations we observed also required 30%-154% less calculation for the Strassen implementations versus Cauchy and Intel MKL DGEMM subroutine methods. The reduction in the number of floating-point operations delivers a performance improvement versus brut-forces algorithm.

The delivered performance in Mega floating-point per second for all the algorithms is presented in Fig.1. This illustrates the overall performance of the best threshold for the Strassen implementation versus the brute-force algorithm and Intel MKL DGEMM subroutine. As we can see Intel MKL DGEMM subroutine for matrix 32768 *x* 32768 achieved 267093 MFLOPS more than the Cauchy brute force algorithm and this difference is close to 19 times. The Strassen algorithm implementation show 17 times weaker MFLOPS results than Intel MKL DGEMM subroutine implementations but more 10% better than Cauchy implementation.

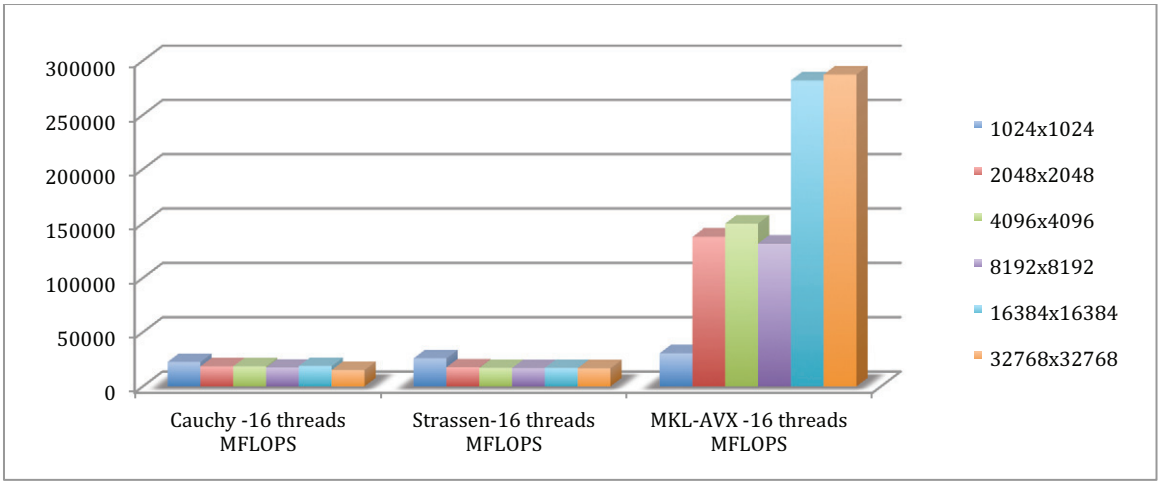

Fig. 1. Achieved performance for all 3 algorithms MFLOPS

5. Effective Method for Matrix Multiplications

The intention of this section of work was to find an effective method for dense matrix-multiplications to overcome the performance delivered by the Intel MKL. Definitely the best implementation for the DGEMM calculation is the Intel MKL DGEMM subroutine with AVX support.

The Intel MKL DGEMM subroutine optimization is closely related to instruction set and hardware architecture. However, the idea is adaptive to other CPU hardware vendors and performance critical libraries as well.

There are several possibilities for improving the performance on DGEMM and other dense linear algebra operations, and Intel MKL DGEMM subroutine implements most of them. The Intel MKL DGEMM subroutine optimization is achieved by various tuning techniques:

• Increasing the memory bandwidth. Blocking is an efficient way to reduce the bandwidth requirement. However, larger blocking puts more pressure on the register file. Consequently, an increase in shared memory bandwidth will require lower blocking factors in registers. The Intel Xeon E5-2680 CPU increases internal cache bandwidth by 800% and memory bandwidth by 200% the Intel MKL DGEMM subroutine consequently utilizes this enhancement.

• Efficient instruction scheduler. Aggressive instruction scheduling improves performance. However, we are limited to instruction ordering and the Intel Xeon E5-2680 hardware scheduler. The actual instructions that are executed depend largely on the hardware scheduling. Therefore, there may be room for improvement by more efficient software scheduling optimized by hand.

• Increasing instruction throughput. This is the ability to perform memory and math operations simultaneously and allow algorithms to reach close to the theoretical peak performance. This is particularly important for AVX and consequently is the area of the biggest performance improvement for the Intel MKL DGEMM subroutine. Optimizations focused on concurrent use of wider 256 bit instructions, is the major attribute of performance delivered by the Intel MKL DGEMM subroutine.

Assuming intensive optimization was done by the Intel MKL team on adopting these 3 techniques our approach was based on the idea to combine the Strassen algorithm with the Intel MKL DGEMM subroutine. We implemented a special hybrid approach based on the Strassen recursive method until optimal threshold is reached and then instead of using the typical brute-force algorithm we used threaded Intel MKL DGEMM subroutines. Before reaching the threshold, the Strassen algorithm performs many matrix additions/subtractions. We used our own routines which can work using offsets from the original matrices, in order to allocate less temporary matrices and thus save memory and reduce memory copies. Additions are implemented using brute-force with OpenMP API.

We have experimented with different thresholds for the sub-matrices and finally found that for the first 3 tested matrices the best results we get are when the threshold is 1024 x 1024. For larger matrices starting from 8192 x 8192 the optimal results are achieved with a threshold of 4096 x 4096. This multithreaded version of code for the first 3

tested sizes of the matrices has an optimal threshold set on the same level as the single threaded implementation as indicated by previous work [4].

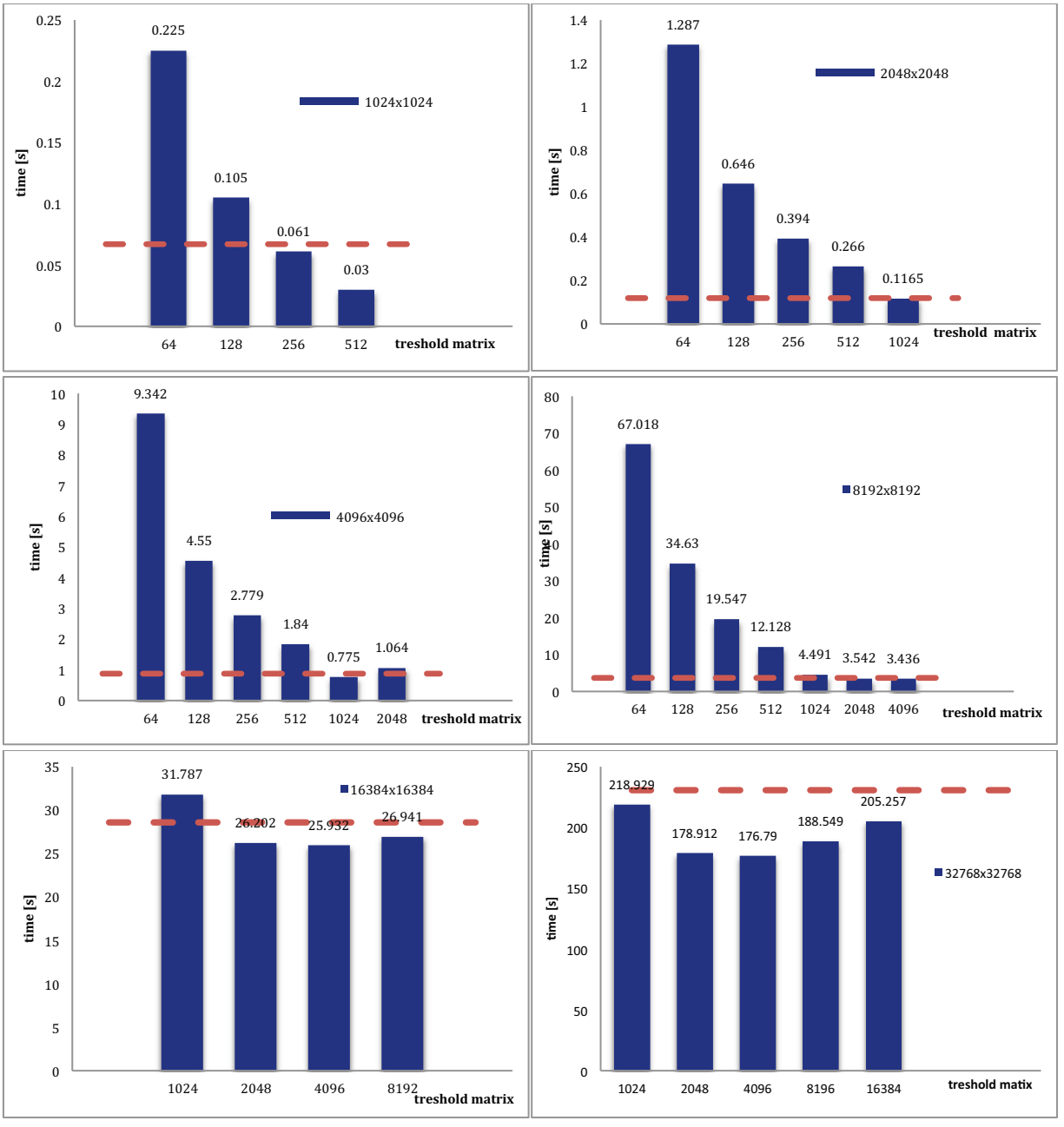

Fig. 2. Execution time of hybrid method versus Intel MKL DGEMM subroutine

Figure 2 presents the execution time of the hybrid method for all tested matrices and for the different sub-matrix thresholds versus the implementations based on ordinary Intel MKL DGEMM subroutine.

For all tested matrix sizes the hybrid method outperforms the best results delivered by the ordinary Intel MKL DGEMM subroutine. The difference is greater when the size of tested matrices increases. The optimal threshold for

sub-matrix size for 3 first tested cases was 1024 *x* 1024. For the larger matrices the optimal threshold for sub-matrix size was 4096 *x* 4096.

For the smallest tested matrix we observed an improvement of 230%, unfortunately for 2048 *x* 2048 the improvement is only 2% and for 4096 *x* 4096 is 12%. For the larger 3 tested cases where the optimal threshold sub-matrix size is 4096 *x* 4096 we see for 8192 *x* 8192 more then 6% improvement, for 16384 *x* 16384 matrix this is 10% and for the 32768 *x* 32768 matrix is a 33% improvement. The proposed hybrid method is particularly effective for the last two tested cases when for almost every size of the sub-matrix thresholds the achieved results are better than results delivered by the ordinary Intel MKL DGEMM subroutine.

6. Conclusion

This paper examined two algorithms for dense matrix-multiplications running on the dual socket platform based on Intel Xeon E5-2680 processor utilizing all 16 threads. In the evaluation study we have tested classic brute-force algorithm and compared it against the Strassen algorithm. Both algorithms have been evaluated against the Intel Math Kernel Library DGEMM subroutine. We have also investigated the number of floating-point MUL and ADD operations, usage of the memory and the achieved performance.

The completed results clearly show that Intel Math Kernel Library with AVX support provides 7 to 19 times better performance versus brute-force algorithm, and 360%-680% versus the Strassen algorithm

The Strassen algorithm implementation proved a performance advantage over the Cauchy implementation and performs 70%-270% faster. The difference is greater as the size of matrix extends. The optimal threshold size for sub-matrix partitioning is 256 *x* 256, we also noticed that for the threshold size of sub-matrix 64 *x* 64 or lower the Strassen algorithm is not effective. The lowlight of the Strassen implementation is 90%-99% higher memory allocation but for 200% better speedup it is a reasonable cost of resources usage.

We have presented the fastest implementation of DGEMM on a dual socket platform utilizing Intel Xeon E5-2680 processor. The optimized DGEMM routine accomplishes 2%-33% better results than the peak speeds attained by Intel MKL DGEMM subroutine. The performance boost is achieved using carefully chosen optimizations based on effective algorithm implementation combined with the latest version of Intel MKL library optimized for the tested hardware.

The proposed hybrid method is based on using the Strassen recursive algorithm until optimal threshold is reached and then use the Intel MKL DGEMM subroutine. This practice delivers 2%-33% better results than the pure Intel MKL DGEMM subroutine. If we consider that for the matrix size of32768 *x* 32768 the classic brute-force algorithm based method completes operations in 4480.9 seconds and the hybrid method can perform all calculations in 176.8 seconds then it looks likely that the proposed solution is very effective.

In future work we are aiming to improve the efficiency of our multi-core architecture, use our instruction set capability more effectively as well as port this to Intel Multicore Architecture (MIC). Using the algorithm we have implemented we hope to further improve the speedup of our proposed hybrid method on the MIC architecture.

Acknowledgements

We gratefully acknowledge the help and support provided by Jamie Wilcox from Intel EMEA Technical Marketing HPC Lab.

References

1. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw., 16:1–17, March 1990.
2. J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. In Proceedings of the IEEE, volume 93, pages 293–312, 2005.
3. K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw., 34:12:1–12:25, May 2008.

4. P.Gepner, V Gamayunov, D. L. Fraser. Evaluation of Executing DGEMM Algorithms on modern MultiCore CPU. In Proceedings of The Parallel and Distributed Computing and Systems 2011 Conference, Dallas, December 2011.
5. Intel® Advanced Vector Extensions Programming Reference, July, 2009
6. L. Gwennap. Sandy Bridge spans generations, Microprocessor Report, September, 2010.
7. P.Gepner, V Gamayunov, D. L. Fraser. Early performance evaluation of AVX for HPC. In Proceedings of International Conference on Computational Science, ICCS 2011, Singapur, June 2011
8. P. Kopta, M. Kulczewski, K. Kurowski, T. Piontek, P. Gepner, M. Puchalski, J. Komasa, Parallel application benchmarks and performance evaluation of the Intel Xeon 7500 family processors, Procedia Computer Science 4 (2011) 372–381.
9. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw., 16:1–17, March 1990
10. V. Strassen. Gaussian elimination is not optimal. Numer. Math., 13:354–356, 1969.
11. G. Henry. Optimize for Intel AVX Using Intel Math Kernel Library's Basic Linear Algebra Subprograms (BLAS) with DGEMM Routine. Intel Software Network, July 2009
12. J. Dongarra., P. Luszczek,A. Petitet,. Linpack Benchmark: Past, Present, and Future, http://www.cs.utk.edu/~luszczek/articles/hplpaper.pdf
13. D. Bailey, E, Barszcz, J.Barton, D. Browning, R.Carter, L.Dagum, R. Fatoohi,S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon,V. Venkatakrishnan,S. Weeratungal, The NAS Parallel Benchamrks, 1994.
14. HPC Challenge Benchmarks, http://icl.cs.utk.edu/hpcc/
15. S. Saini, D. Talcott, D. Jespersen, J. Djomehri, H. Jin, and R.Biswas, Scientific Application-based Performance Comparison of SGI Altix 4700, IBM Power5+, and SGI ICE 8200 Supercomputers, Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Austin, Texas, 2008.